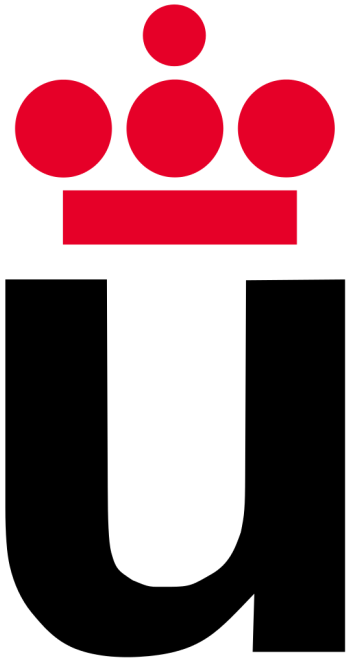
Desarrollo de juegos con inteligencia artificial

Práctica 2: Machine Learning



21/06/2024

Luis Antonio González Martínez
Sergio Jesús González Castilla

Índice de contenidos

Índice de figuras

1. Introducción: Contexto de la práctica.

Para el desarrollo de la segunda práctica será necesario un poco de su contextualización, el objetivo de la misma será la de implementar un agente inteligente (en este caso, nuestro zombie principal del entorno), cuyo comportamiento sea aprendido por técnicas de machine learning, concretamente, se emplearán técnicas de Q-Learning para entrenar el cerebro del agente (su IQ). Para resolver este problema, se utilizará uno de los motores de videojuegos más comunes de nuestro sector "Unity", y su lenguaje de programación "C#.

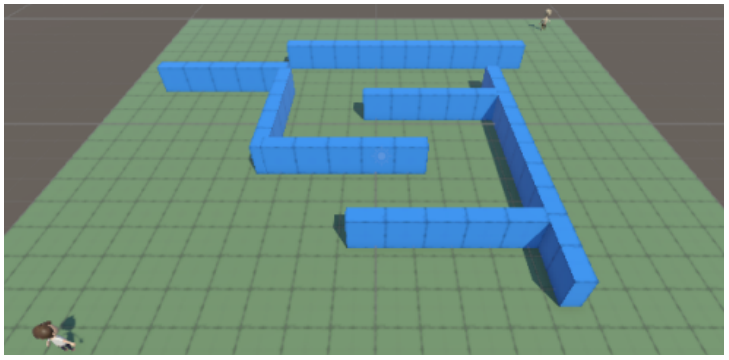


Figura 1: Agente inteligente, su enemigo y escena de entrenamiento.

2. Problema que se le plantea a la IA.

El problema planteado a resolver no es ni más ni menos que utilizar nuestra escena base "TrainPlayGround", para realizar el entrenamiento de nuestro Agente inteligente, teniendo en cuenta a todos los elementos de la escena:

- Game object Agent = agente inteligente controlado por la IA
- Rival que se desplaza por el escenario en búsqueda del agente inteligente mediante un algoritmo de A*
- QMindTrainer contiene el script que entrena al agente inteligente
- El escenario (gameobject Scenary), que posee "Floor y Wall" para dar la información de poder ser transitado o de obstáculo.

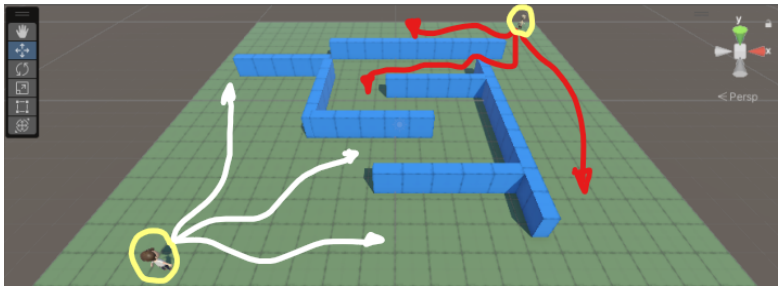


Figura 2: Agente inteligente y Enemigo en movimiento.

3. Algoritmo Q-learning.

Para el desarrollo pleno de la práctica se empleará en su totalidad el algoritmo **Q-Learning**. El mismo hará que nuestro *agente inteligente* trate de **no ser atrapado** por el *jugador*. En esta práctica, el agente inteligente será el zombie, mientras que el jugador será el humano. El algoritmo, sigue el **patrón del aprendizaje por refuerzo**, es decir, se le dará *recompensa* al agente inteligente siempre y cuando su *acción sea buena* para su situación, mientras que si realiza acciones negativas, se le penalizará.

Se realiza *interacción entre el agente y su entorno*, para poder definir las **acciones** a plantear, y los **posibles estados** a los que podrá pasar (siendo estos lo más positivos posibles, conseguidos gracias a un entrenamiento de la IA).Todo lo mencionado definirá la cadena de estados hasta el fin de la ejecución del Script, teniendo como respuesta el número total de estados por los que ha pasado el agente inteligente.

Centrándonos en el estado, lo podemos definir como una *instancia de tiempo dentro de todo el entorno*, cada vez que el agente realiza una acción, esté siempre cambiará, dando paso al siguiente. Todos **los elementos** a tener en cuenta para el cambio de estado (por ejemplo, en nuestro caso, las direcciones, distancia y orientaciones) **definen el mismo**, y *solo* se pueden medir los elementos elegidos para su aprendizaje.

Todo el proceso de aprendizaje, se guarda en la denominada **Tabla Q**, dentro de la misma se contienen todos los conocimientos que tiene o va adquiriendo el agente, todos *sus valores representan la recompensa total* obtenida al ejecutar cada una de las acciones. Las tablas se crean durante el proceso de entrenamiento, para después poder el agente desempeñar su objetivo de la forma más llevadera en un escenario real.

Los **parámetros** que se miden dentro del algoritmo son tres, el primero de ellos es el **learning rate**, define cuánto aprende nuestro algoritmo con cada simulación, el segundo es el **discount factor**, el mismo parte de la recompensa futura que tomamos, por último, el **exploration rate**, nos comenta cuanto exploramos nuevas acciones. Centrándonos en el último mencionado, siendo este uno de los más importantes, es la función encargada para *seleccionar las acciones* que se realizan en cada estado.

Algoritmo Q-Learning greedy

```
1: procedure Q-LEARNING
2:     α ← learning rate
3:     γ ← discount factor
4:     ϵ ← exploration rate
5:     for each episode do
6:         s ← randomState
7:         repeat
8:             a ← selectAction possibleActions, ϵ
9:             updateWithReward Q(s, a), α, γ
10:        until terminal s
11:    end for
12: end procedure
```

Figura 3: Pseudocódigo del algoritmo.

El cálculo de los nuevos estados se realiza mediante una regla de aprendizaje, la misma viene definida por la ecuación planteada a continuación:

$$Q'^{(s,a)} = (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{\forall a'} Q(s',a'))$$

Figura 4: Ecuación cálculo de estados.

El significado de cada uno de los elementos de la ecuación son las siguientes:

- $\boldsymbol{Q'(s,a)}$ es el nuevo valor $\boldsymbol{Q}$ en el estado $\boldsymbol{s}$ realizando la acción $\boldsymbol{a}$
- $\boldsymbol{(1-\alpha)Q(s,a)}$ es la fracción del valor $\boldsymbol{Q}$ anterior que mantenemos
- $\boldsymbol{\alpha(r + \gamma \max_{\forall a} Q(s',a))}$ es la fracción que "aprendemos", que se calcula con:
  - $\boldsymbol{r}$ es la ***recompensa*** recibida por esta acción
  - $\boldsymbol{\gamma \max_{\forall a'} Q(s',a')}$ es una fracción de la recompensa que *obtendremos* en la ***siguiente acción***. Mejorará la recompensa en función de cuanto nos acerca a la meta esta acción.

Figura 5: Ecuación cálculo de estados (información).

4. Procedimiento de desarrollo.

Como buena práctica, lo primero que se ha llevado a cabo ha sido la declaración del Modelo, es decir, se han definido los diferentes elementos a considerar para su completo desarrollo.

Primero de todo, se ha definido el estado (QState, clase embebida en QTable), para así contemplar todo lo que el personaje sabe desde su base (direcciones posibles, distancia y orientaciones). Como segundo punto, tenemos la definición de la tabla "Q"(QTable), la misma es el cerebro del agente, dentro de ella se almacena la información referente a los estados y recompensas del agente, es la herramienta que se usa para que vaya aprendiendo, gracias a la información del mundo proporcionada. Por último, es necesario definir la recompensa, para así contemplar si el agente está realizando de forma correcta la tarea encomendada. Dentro de nuestra propia implementación, la recompensa negativa será mayor que la positiva, es decir, la penalización por fallo será muy superior a la recompensa por realizar una buena acción.

La programación del estado sirve para definir cómo va a ser la **tabla "Q"** y qué elementos va a tener en cuenta para la actualización de estados. La tabla tiene como estructura la de un diccionario (estructura facilitada por C#), dentro se almacena toda la información referente a la creación, guardado y lectura de estados y gestión de sus valores.

Para que el agente actualice la tabla de manera continua, se le somete a un entrenamiento y luego se guarda su progreso. Para que esta información pueda ser

utilizada para desenvolverse en otro entorno, es necesario tener una **Persistencia de la tabla**. Con la mencionada persistencia conseguimos grabar la tabla y leerla cuando es necesario.

Lo más importante a parte de la tabla, sería el **Trainer**. Este es el que tiene encomendada la tarea de entrenar al agente. Se declaran dos programas, uno de entrenamiento y otro de ejecución del agente. Cuando el proceso de entrenamiento se ejecuta, surge la necesidad de guardar lo que aprende el agente, aquí entra en juego la persistencia comentada anteriormente.

Como buena práctica, a la hora de almacenar información, no está de más fragmentar la tabla en las diferentes épocas (agrupación de estados) para poder así aprender de mejor forma, sometiendo al agente de diferentes formas, y eligiendo la época con mejor recompensa obtenida para seguir creciendo el aprendizaje de esa rama.

5. Código explicado. QTable

Este código es una implementación dedicada a la creación de una **tablaQ**, la cual es utilizada para el **entrenamiento** de un **agente** usando el algoritmo de **Q-Learning**, siendo el mismo un algoritmo muy utilizado, siendo de *aprendizaje por refuerzo*. La clase **QTable** es la encargada de gestionar toda la tabla Q y los estados del agente en el entorno.

La clase **QState** representa el estado del agente en el entorno. Incluye cuatro booleanos para los movimientos posibles (*up,right,down y left*), y dos enteros para la *distancia* y la *orientación* respecto al enemigo. Por otro lado, los constructores **inicializan** la tabla Q y establecen los *parámetros* necesarios para el entrenamiento y la *segmentación* del mundo, Estos métodos, en resumidas cuentas, (UpdateWithReward, DiscretizeDistance, DiscretizeAngle, GetReward, GetState,GetTrainingAction, GetAction y GetAgentMovement) gestionan la actualización de la tabla Q, la discretización de las distancias y ángulos, la obtención de **recompensas** y la determinación de **estados** y **acciones** del agente.

Por último, se realiza Inicialización de una tabla con los estados inicializados y los arrays de valores vacíos (la asignación de un valor a 0.0f por defecto)

6. Código explicado. QTrainer

A Continuación se presenta un resumen sobre el propósito de la clase **QTrainer** y su principal funcionalidad, la misma *hereda* componentes de **IQMindTrainer**, y contiene una de las funcionalidades más importantes de todo el programa, gracias a la misma, el agente es capaz de *aprender*, para cuando se someta a un escenario real, ser capaz de desenvolverse.

La funcionalidad más importante se realiza en “**DoStep**”, en ella lo primero que se realiza es una *comprobación*, para verificar si está **inicializado los estados** de forma correcta, en el caso afirmativo, se realiza un primer proceso de **identificación de estados**, en el que se encuentra el estado inicial y la antigua distancia presentes. Como segundo paso tenemos la **elección** de la **acción** del agente, para pasar a poder moverlo. Trás

realizar el proceso anterior, se pasa a la **elección** de **recompensa**, la misma dependerá de la situación en el estado del personaje, variando entre los diferentes parámetros planteados, otorgando al agente la *recompensa pertinente* (positiva o negativa).

Dentro de **DoStep**, por último se *actualiza la TablaQ* con los valores calculados, y se define si se debe pasar al siguiente episodio o no, dependiendo de si el agente ha sido atrapado o en contraposición, ha realizado alguna acción no permitida.

Al final del script, se localiza “Inizialize()”, la misma recoge información de entrada sobre “QMindTrainerParams, WorldInfo y INavigationAlgorithm” para poder recoger los parametros, ser asignados a variables, y finalmente, ser pasados a inicialización.

7. Proceso de aprendizaje.

Para el desarrollo íntegro de la IA, para que la misma sea capaz de poder solucionar el problema planteado en la práctica, se debe de realizar un proceso de aprendizaje, en el que la máquina *ejecuta numerosas iteraciones* para ir completando poco a poco la TablaQ.

Para un aprendizaje más rico, es necesario *ir modificando ciertos parámetros* (como la recompensa) en diferentes ejecuciones del entrenamiento. En muchas ocasiones, se han *planteado diferentes mapas*, donde la IA debía de intentar desenvolverse, en diferentes situaciones de bajo o alto estrés a diferentes distancias del enemigo, con la intencionalidad de que la misma se adaptara a cualquier situación planteada. Por otro lado, la recompensa, al ser un factor tan crucial en el aprendizaje, dependiendo del mapa y su dificultad, *modificaremos los valores de penalización y bonificación* de la misma, para que vaya poco a poco siendo más consecuente de sus acciones.

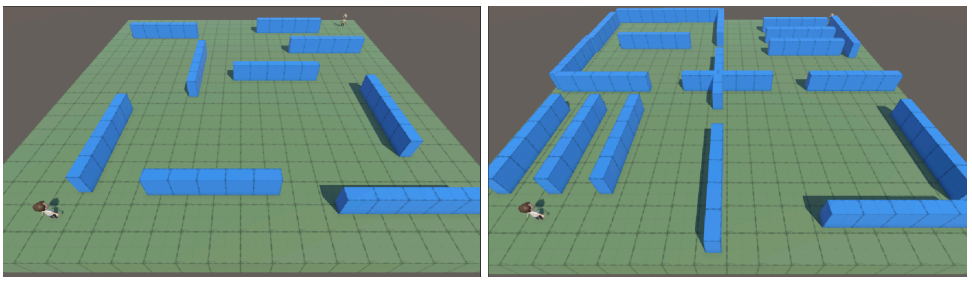


Figura 6: Ejemplos de mapas de aprendizaje.

Otro parámetro a tener en cuenta durante el proceso es “epsilon” o *exploration rate*, el mismo indica el índice de aleatoriedad de acciones que realiza el agente durante su entrenamiento. En un primer contacto, el epsilon tiene un valor alto (0.85) y durante el resto de los siguientes entrenamientos, un valor bajo (0.3). Por otro lado, otros parámetros como *learning rate* o *discount factor* no han sido tan significativos para el resultado final de la tabla.

Este proceso ha ocupado mucho tiempo, provocado al gran número de iteraciones diferentes planteadas durante su desarrollo, además, por organización, se creaban diferentes Tablas Q, para poder así compararlas y observar qué método era el más

apropiado. A Continuación, se observan algunos ejemplos de los diferentes mapas planteados a la IA durante el entrenamiento de la misma.

8. Bibliografía y herramientas.

Bibliografía:

David María Arribas y Dan Casas Guix (Año académico 2023/24).
Aprendizaje por refuerzo [presentación presente en la asignatura Desarrollo de juegos con inteligencia artificial].

Laura Llopis Ibor (Año académico 2023/24).
C# cheat sheet [presentación presente en la asignatura Ingeniería de Videojuegos]

Herramientas:

Se ha utilizado el software de Unity y Visual Studio 22, aparte de herramientas de dibujo para representar la ejecución a obtener.